Transaction Level Analysis for a Clustered and Hardware-Enhanced Task Manager on Homogeneous Many-Core Systems

Daniel Gregorek, Robert Schmidt, Alberto García-Ortiz
Institute of Electrodynamics and Microelectronics, University of Bremen, Germany
{gregorek,agarcia}@item.uni-bremen.de, r.schmidt@uni-bremen.de

ABSTRACT

The increasing parallelism of many-core systems demands for efficient strategies for the run-time system management. Due to the large number of cores the management overhead has a rising impact to the overall system performance. This work analyzes a clustered infrastructure of dedicated hardware nodes to manage a homogeneous many-core system. The hardware nodes implement a message passing protocol and perform the task mapping and synchronization at run-time. To make meaningful mapping decisions, the global management nodes employ a workload status communication mechanism.

This paper discusses the design-space of the dedicated infrastructure by means of task mapping use-cases and a parallel benchmark including application-interference. We evaluate the architecture in terms of application speedup and analyze the mechanism for the status communication. A comparison versus centralized and fully-distributed configurations demonstrates the reduction of the computation and communication management overhead for our approach.

General Terms

design, architecture

Keywords

many-core, embedded system, run-time management, message passing, task mapping, dedicated hardware

1. INTRODUCTION

Power-efficiency and scalability has been a driver for a variety of cluster-based many-core systems. Among them, the P2012 (a.k.a. STHORM) many-core architecture, the MPPA manycore and the Single-Chip Cloud Computer SCC have recently been implemented as real-world hardware instances [1] [6] [11]. Their designs address power budgets ranging from 2W to 125W and incorporate a multitude of architectural features and programming models.

The domain of many-cores leads to the demand for a sophisticated (re-)design of the run-time task management. The task management has to bring the dynamic requirements of the user applications into accordance with the monitored state of the chip. Also, a task manager is responsible for allocating the resources (1) computation, (2) communication and (3) memory to the applications. Hardware-assistance has become a key factor to reduce the overhead introduced by the run-time task manager [19].

The idea of hardware task scheduling can be tracked back to the POLYP mainframe computer [18]. An overview about separate task synchronization subsystems is given by Herkersdorf [10]. But to our best knowledge, we are the first to present and to analyze a *full-fledged* on-chip task management infrastructure using a dedicated infrastructure of hardware nodes. Key objective of the dedicated infrastructure is to conceal the resulting management overhead from the user tasks.

The remainder of this paper is organized as follows: Section 2 discusses related work. In Section 3 we present our proposed architecture and Section 4 introduces the run-time task manager. Section 5 shows experimental results, and finally Section 6 concludes the paper.

2. RELATED WORK

A hardware-assisted run-time software for embedded many-cores is presented by HARS [17]. But, while using the hardware semaphores included in the STHORM many-core architecture their evaluation is limited to intra-cluster task synchronization.

A distributed run-time is proposed for the MPPA [7]. The run-time environment exploits a dedicated system core which acts as a resource manager inside a single cluster. However, their approach is constrained to a compile-time (static) mapping scheme.

The SCC comes with a default Linux configuration and the message passing programming model. Also, basic synchronization primitives are implemented in hardware [20]. The SCC consists of small-size clusters which yet not contain a dedicated management core.

Besides clustered solutions there exist centralized as well as fully-distributed approaches. Nexus++ uses a single application specific circuit resolving time-critical task dependencies at run-time [5] and applies a trace-based description of a H.264 benchmark. A distributed and dedicated hardware approach has been implemented by Isonet [15]. Isonet applies a fully-distributed network of dedicated management nodes for hardware supported load balancing.

3. SYSTEM ARCHITECTURE

This paper is a continuation of our work presented in [8] and analyzes a clustered architecture for the task management. Our overall system architecture is constructed by a *homo-*

geneous many-core system as a baseline which is enhanced by a dedicated management infrastructure. The dedicated management infrastructure is implemented as a network of global management nodes and clusters of local controllers. Each local controller is tightly coupled to a processing element. The global management nodes are connected to a global interconnect. A local interconnect links one global management node with its local controllers. Fig. 1 gives an outline of the proposed architecture. The interconnects are implemented as (but not restricted to) shared buses. A common interconnect between the processing elements is left out for better readability. The communication between the dedicated nodes is done by means of message passing. Each node contains message queues for transmission and reception.

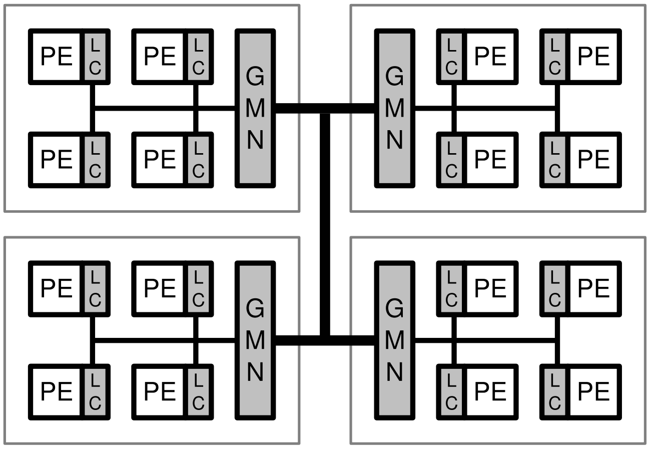


Figure 1: Outline of the system architecture for the clustered task management. Having $k = 4$ global management nodes (GMN) and $m = 16$ local controllers (LC) coupled to the processing elements.

3.1 Global Management Nodes

Each of the global management nodes runs one instance of the run-time task manager in software and contains dedicated hardware for message processing. The communication between the nodes is determined by the message protocol explained in Sec. 3.3. The execution of the system-calls from the user tasks is realized by the global nodes. Additionally, they implement a hierarchical task mapping algorithm and a cluster status communication mechanism at run-time (see Sec. 4).

The global management nodes demand for programmability and for a minimal area footprint. Messaging between the nodes requires for fast interrupt handling. We plan to implement the global management nodes by means of programmable stack machines. Stack machines have very-low hardware complexity [16], exhibit high performance in subroutine calls (context switching), and achieve deterministic time for interrupt handling [13] [12]. Another advantage is the small code size of programs written for stack machines [3]. Small memory footprints allow to spend each global node its own program memory, which diminishes communication overhead for instruction fetching.

3.2 Local Controller

A local controller (LC) is tightly coupled to a processing element (PE) for user task execution. The PE contains a functional model of a RISC-like processor architecture and executes a trace-based description language. The traces are used to raise the system-calls given in Tab. 2 and determine the application behavior (see Sec. 5).

The local controller maintains a system call dispatcher for low-latency response and has access to the PE registers. The dedicated LC can be implemented with low area overhead [9] and operates in parallel to the PE. Any system-call from a user task is fetched by the LC and forwarded to its global node by means of a dedicated message. Due to the dedicated infrastructure for the task management the PE does only execute the user tasks.

3.3 Messaging Protocol

We send messages via the dedicated interconnects for communication between the hardware nodes. The message passing combines data transport and run-time system synchronization. Each message has a header and one or more 32-Bit data fields. Table 1 displays the structure of a message. The header contains the message type, at least the source address, the priority and a broadcast flag. The size of the message header depends on the actual hardware configuration (i.e. number of nodes / address-width).

Table 1: Message structure

`type`	`src`	`dst`	`prio`	`flag`	`data`

Most of the message types directly correspond to the system calls given in Tab. 2 and are send from a local controller to its global node. Beyond that, a message `task-start` invokes the start of a task. That message transports the address of the `task-control-block` (see [14]) and the `stack-pointer` as message data, and can be send from a global node to a local controller as well as to another global node. Further, the global nodes use the message `status-beacon` to broadcast the current workload status (see Sec. 4.2) to all other global nodes.

4. TASK MANAGER

The task manager we use is loosely based on the Micro-C/OS-II [14] software operating system. We do not adapt real-time capabilities but extended the task manager to have basic multi-core functionality. The extensions to the task manager are explained in Sec. 4.1 and 4.2. We replaced the task scheduler to employ a simple first-come-first-serve strategy.

The system-calls, which we apply throughout this paper are explained in Tab. 2. We use a customized join/barrier mechanism to synchronize the user tasks. To reduce the number of system-calls, a child task is allowed to exit immediately, when signalizing a `join-exit`.

4.1 Task Mapping

The task mapping algorithm is part of the software OS and is implemented inside every global node. Since our targeted task scheduling problems consist of task sets having a large

Table 2: System calls

Name	Param.	Description
`rcsv-spwn`	`imem`, `dmem`, `cnt`	Spawn new recursive task of given count (`cnt`) and instruction- (`imem`) and data (`dmem`) memory addresses
`rcsv-exit`	`addr`	Terminate task
`join-init`	`cnt`	Initialize join barrier with given initial count and return its address to user
`join-free`	`addr`	Free join barrier from memory
`join-wait`	`addr`	Let task wait until counter is zero
`join-exit`	`addr`	Decrement counter and terminate task

number of tasks, we use a recursive task spawning/fork strategy. Every recursive task spawns two additional helper task and then blocks until its child's have terminated. The recursion is executed until one of the following stop conditions is reached:

1. The number of remaining child tasks is smaller than or equal to the number of PEs per cluster
2. The number of active helper tasks is greater than or equal to the number of clusters

The recursive start-up follows a dynamic cluster mapping procedure, which tries to equally distribute the recursive helper tasks onto the clusters. After the binary fork-tree has stopped to expand, the actual child tasks of the application are spawned. This final number of working child tasks is fixed and determined by the application profile.

The mapping problem is therefore *split* into two stages: At the first stage, the mapping algorithm is responsible for selecting the global nodes (clusters), where the helper tasks get mapped to. At the second stage, the mapping algorithm selects the local processing elements, where the actual child tasks get mapped to. Each single mapping decision is done by means of a min-search. The mapping algorithm chooses that node with the minimal number of mapped tasks. To do this, every global node maintains a data structure about the per-PE workload inside his private cluster and a data structure about the summarized workload for each remote global node. In the current implementation, we estimate the workload by counting the total sum of locally mapped tasks.

Mapping is done only once, we do not allow a task to restart at any different location (run-time migration), since these operations usually come at a high performance penalty [2] and are not in the focus of our analysis.

4.2 Status communication

Communicating the workload status is required for allowing the mapping algorithm to make meaningful decisions. Due to the shared nature of the global bus interconnect we use a broadcast message to inform all collaborating nodes about the local workload. We implemented a threshold-based mechanisms for broadcasting the total sum of locally mapped tasks. The mechanism triggers a broadcast, every time a certain threshold Δn_{th} in change of the number of mapped tasks is reached.

5. EVALUATION

We employ a simplified task-based programming model for our analysis. Parent tasks may spawn numerous child tasks and wait until their computation has finished. Our main criterion for evaluation is the throughput time t_r (response time) of the overall application (parent + childs). We measure the speedup as the ratio of the sequential throughput time $t_{r,seq}$ vs. the achievable parallel throughput time $t_{r,par}$ and show that the achievable speedup $S = t_{r,seq}/t_{r,par}$ is either limited by the computation or communication management overhead.

5.1 Analytic Model:

Having n independent child tasks of equal length l, m homogeneous processing elements and k global management nodes the maximal achievable speedup is limited by a temporal management overhead $\Omega(m,n,k)$ as shown in Eqn. (1):

$$S = \frac{t_{r,seq}}{t_{r,par}} = \frac{n \cdot l}{t_{r,par}} = \frac{n \cdot l}{\lceil n/m \rceil \cdot l + \Omega(m,n,k)} \tag{1}$$

Due to the considered run-time computation of the mapping problem there is a computation overhead Ω_{cmp}. Having multiple global nodes k there is an overhead Ω_{msg} in communication. We constitute the overall management overhead Ω depending on the number of processing elements m, the number of user tasks n and the number of global management nodes k by equation (2):

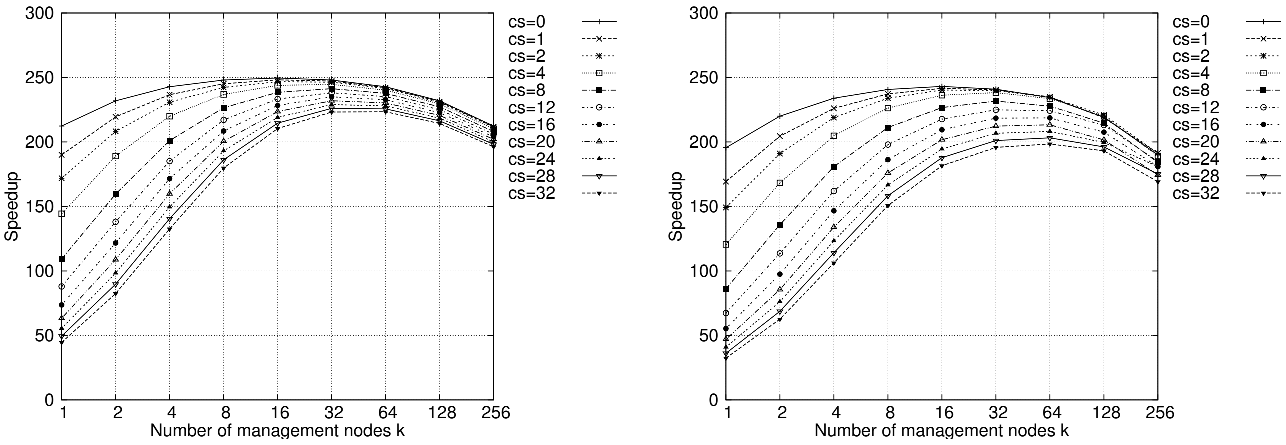
$$\Omega(m,n,k) = \Omega_{cmp}(m,n,k) + \Omega_{msg}(m,n,k) \tag{2}$$

Each decision of our task mapping algorithm (See Sec. 4.1) infers a selection time overhead Ω_s. Due to the recursive task startup there is a logarithmic dependency ($\log n$) for the global mapping stage. The resulting overhead Ω_{cmp} for computing the mapping problem of n user tasks is given by equation (3):

$$\Omega_{cmp}(m,n,k) = \overbrace{\log(n) \cdot \Omega_s(k)}^{map\ global} + \overbrace{\frac{n}{k} \cdot \Omega_s\left(\frac{m}{k}\right)}^{map\ local} \tag{3}$$

The required search function for the mapping algorithm can be implemented having logarithmic time-complexity $\mathcal{O}(\log \nu)$ by e.g. Red-Black Trees [4]. The selection time Ω_s for one decision of the mapping is modeled as $\Omega_s = c_s \cdot \log \nu$; where ν is the number of nodes to be searched through and c_s is a timing parameter of our framework (see Tab. 3). Correspondingly, the communication overhead due to intra- and inter-cluster messaging is approximated by means of Eqn. 4:

$$\Omega_{msg}(m,n,k) = \overbrace{c_b \cdot k}^{global} + \overbrace{c_b \cdot \frac{m}{k}}^{local} \tag{4}$$



(a) Analytic model for the speedup using the recursive task startup. Having $m = 256$ PEs and $n = 256$ child tasks for a varying number of global nodes k and coefficient c_s

(b) Measured result for the speedup using the recursive task startup. Having $m = 256$ PEs and $n = 256$ child tasks for varying global nodes k and the delay coefficient c_s

Figure 2: Independent tasks on 256 homogeneous processing elements

Table 3: Default parameters for the analytic evaluation and the transaction level simulations

Name	Value
Number of processing elements	256
Global bus width	32 bit
Local bus width	32 bit
Message receive delay ($c_b/2$)	4 Ticks
Message transmit delay ($c_b/2$)	4 Ticks
Selection delay coefficient (c_s)	8 Ticks
Max. child task length	16000 Ticks
Simulation length	1e7 Ticks

Eqn. 4 introduces the timing parameter c_b to model the time delay inquired by communication messages. In Fig. 2a the projected speedup is plotted for the analytic model. We set $m = 256$ PEs and $n = 256$ child tasks while varying the number of global nodes and the coefficient c_s. As indicated, the recursive startup and task mapping favors a number of $32 - 64$ global management nodes.

5.2 Experimental Setup

We use the transaction-level simulator presented in [8] to evaluate our architecture and to compare the analytic model against the simulation result. Table 3 gives the default parameters for our model. Our evaluation ignores wire capacitances, which factual privileges fully-centralized or fully-distributed configurations with a large number of nodes attached to the local or global interconnects. To eliminate the effect of bottlenecks at the interconnects we previously analyzed and set the bit-width of the buses to a convenient value of 32 bit.

5.3 Independent Tasks

The benchmarks are modeled by means of a trace description language. The traces describe the computation and memory access patterns of the tasks as well as the calls to the run-time services (system calls). The traces are interpreted and executed by the model for the processing elements. In our current analysis we use a synthetic parallel benchmark consisting of n independent tasks without any memory access. Fig. 2b shows the measured speedup fitting quite well to the analytic description due to the regular nature of the benchmark.

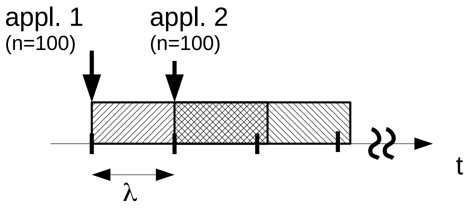
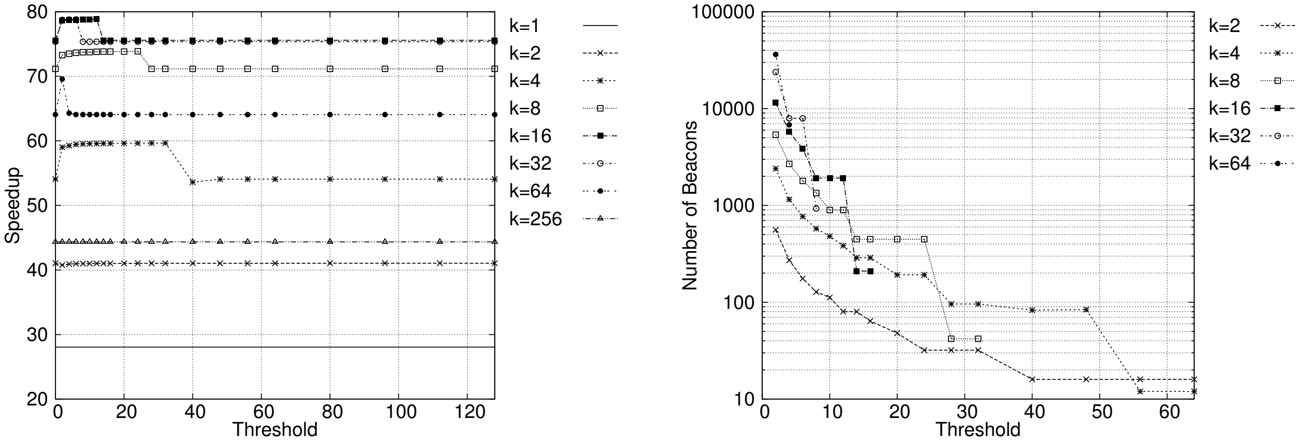


Figure 4: Periodic start-up sequence for two competing applications with inter-arrival time λ and n child tasks

5.4 Application Interference

In the second experiment we included interference between two competing applications having equal priority. The application start-up sequence with inter-arrival time λ as shown in Fig. 4 is repeated periodically. The inter-arrival time λ is Poisson distributed and has a mean value of $\lambda = 7999$ Ticks. The number of processing elements is $m = 256$ and each application has $n = 100$ child tasks. The child task length has a uniform distribution between 95 - 100 % of the maximum computation time. The synchronization between the parent and the child tasks is done by means of the fork/join mechanism presented in Sec. 4. The stimulus is active for 90 % of the simulation time and is send with highest priority directly to a randomly chosen global node. The other global nodes are kept *agnostic* about arriving applications and must update their information according to the presented status communication (see Sec. 4.2). We do not display any values, where the number of completed applications differs from the number of injected ones (no misses are allowed).



(a) Evaluation of the speedup versus the threshold Δn_{th} for the threshold-based workload status communication mechanism having different numbers of global nodes

(b) Total number of transmitted beacons for workload status communication versus the threshold Δn_{th} having different numbers of global nodes

Figure 3: Application interference on 256 homogeneous processing elements

Fig. 3a shows the resulting application speedup for the hierarchical task mapping algorithm (see Sec. 4) and the threshold-based status communication mechanism. Using $k = 16$ global hardware nodes and a threshold $\Delta n_{th} = 4$ a speedup improvement by a factor of 2.8 compared to $k = 1$ is achieved. For a fully-distributed configuration the improvement factor is only around 1.6 compared to $k = 1$. Using the given benchmark, the threshold based mechanism reveals a robust load balancing as long as the threshold is smaller than the number of processing elements per cluster.

Further, we display the number of transmitted status beacons for the threshold-based mechanism in Fig. 3b. The figure gives an indicator about the required energy for status communication, which is related to the number of *received* beacons. Every transmitted beacon must be received by all remote nodes to fully synchronize the network. For a threshold of $\Delta n_{th} = 4$ it is indicated that a fine-grained clustered configuration with $k = 32$ management nodes must transmit an amount of beacons that is around 1.37 higher compared to a configuration having $k = 16$ nodes.

For a preliminary area analysis we compare an in-house implementation of a dedicated 32-Bit stack machine as global management node (GMN) to an mLite/PLASMA CPU [21] as processing element. Both designs have been synthesized using an industrial 65nm low-power technology (see Tab. 4). When disregarding an additional multiplier having $3547 \mu m^2$ shipped inside the mLite, we still can report around 25% less area for the stack machine.

Table 4: Synthesis results for 65nm low-power

Unit	Comb. $[\mu m^2]$	Non-comb. $[\mu m^2]$	T_{clk}
GMN	9290.4	9881.2	1.77 ns
mLite	16268.4	12909.5	1.79 ns

5.5 Summary

In Tab. 5 we summarize the results of our evaluation in terms of application speedup using the presented hardware infrastructure. As a comparison we give our obtained values for a fully-centralized configuration (like e.g. Nexus++ [5]) and a fully-distributed one (like e.g. Isonet [15]). The table indicates the significant impact of the management overhead, which was constituted by Eqn. (2), (3) and (4). As a further work, we plan to consider a cycle-accurate model of the task manager and analyze the overall power consumption of the system. To get a more realistic scenario about the user applications, their memory access will be considered as well.

Table 5: Comparison of Speedup ($S = t_{seq}/t_{par}$) for $n = 100$ independent tasks on $m = 256$ PEs using different numbers of cluster (global nodes) k.

k	Speedup	Ref.
1	28.1	Centralized, like e.g. Nexus++ [5]
8	73.5	this work
16	78.7	this work
256	44.3	Distributed, like e.g. Isonet [15]

6. CONCLUSION

A dedicated infrastructure of hardware nodes for run-time task management has been introduced. Compared to previous works we consider a full-fledged and separated task management infrastructure. The infrastructure uses a message passing protocol and allows a design trade-off between the advantages of centralized and fully-distributed architectures by choosing an optimal cluster size.

We analyze the clustered architecture by means of an analytic description as well as by transaction level simulations using a parallel benchmark including application interference. Our simulations revealed significant impact of the management overhead to the overall system performance.

The management overhead for the task mapping problem can be reduced by using our infrastructure and a two-stage task mapping approach. Having $m = 256$ processing elements and choosing the optimal cluster size can provide a performance improvement by a factor of 2.8 compared to a single-cluster/centralized configuration.

The results further show the dependency of the run-time management system on the status information from remote clusters. The lack of information may lead to inappropriate mapping decisions causing a performance drawback. We measured the communication overhead by counting the number of status beacons transmitted by the global management nodes. Using a threshold-based mechanism for status communication and the optimal cluster size, we measured a significant reduction in terms of transmitted synchronization messages compared to more fine-grained clustered configurations.

7. REFERENCES

[1] L. Benini, E. Flamand, D. Fuin, and D. Melpignano. P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 983–987. EDA Consortium, 2012.

[2] S. Bertozzi, A. Acquaviva, D. Bertozzi, and A. Poggiali. Supporting task migration in multi-processor systems-on-chip: a feasibility study. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, pages 15–20. European Design and Automation Association, 2006.

[3] J. Bowman. J1: a small Forth CPU Core for FPGAs. In *EuroForth 2010 Conference Proceedings*, pages 43–46, 2010.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, et al. *Introduction to algorithms*, volume 2. MIT press Cambridge, 2001.

[5] T. Dallou and B. Juurlink. Hardware-based task dependency resolution for the starss programming model. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*. IEEE, 2012.

[6] B. D. de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P. G. de Massas, F. Jacquet, S. Jones, N. M. Chaisemartin, F. Riss, et al. A clustered manycore processor architecture for embedded and accelerated applications. In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pages 1–6. IEEE, 2013.

[7] B. D. d. Dinechin, P. G. d. Massas, G. Lager, C. Léger, B. Orgogozo, J. Reybert, and T. Strudel. A distributed run-time environment for the kalray mppa-256 integrated manycore processor. *Procedia Computer Science*, 18:1654–1663, 2013.

[8] D. Gregorek and A. Garcia-Ortiz. A transaction-level framework for design-space exploration of hardware-enhanced operating systems. In *International Symposium on System-on-Chip 2014 (SOC) 2014*. IEEE, 2014.

[9] D. Gregorek, C. Osewold, and A. Garcia-Ortiz. A scalable hardware implementation of a best-effort scheduler for multicore processors. In *Digital System Design (DSD), 2013 Euromicro Conference on*, pages 721–727. IEEE, 2013.

[10] A. Herkersdorf, A. Lankes, M. Meitinger, R. Ohlendorf, S. Wallentowitz, T. Wild, and J. Zeppenfeld. Hardware support for efficient resource utilization in manycore processor systems. In *Multiprocessor System-on-Chip*, pages 57–87. Springer, 2011.

[11] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, et al. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 108–109. IEEE, 2010.

[12] P. J. Koopman. Modern Stack Computer Architecture. In *Systems Design & Networks Conference*, pages 153–164, 1990.

[13] P. Koopman Jr. *Stack computers - the new wave.* Mountain View Press, Route 2 Box 429, LaHonda, CA 94020, 1989.

[14] J. J. Labrosse. *Microc/OS-II.* R & D Books, 1998.

[15] J. Lee, C. Nicopoulos, H. G. Lee, S. Panth, S. K. Lim, and J. Kim. Isonet: Hardware-based job queue management for many-core architectures. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 21(6):1080–1093, 2013.

[16] P. H. W. Leong, P. Tsang, and T. Lee. A fpga based forth microprocessor. In *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*, pages 254–255. IEEE, 1998.

[17] Y. Lhuillier, M. Ojail, A. Guerre, J.-M. Philippe, K. B. Chehida, F. Thabet, C. Andriamisaina, C. Jaber, and R. David. Hars: A hardware-assisted runtime software for embedded many-core architectures. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(3s):102, 2014.

[18] R. Manner. Hardware task/processor scheduling in a polyprocessor environment. *Computers, IEEE Transactions on*, 100(7):626–636, 1984.

[19] V. Nollet, D. Verkest, and H. Corporaal. A safari through the mpsoc run-time management jungle. *Journal of Signal Processing Systems*, 60(2):251–268, 2010.

[20] P. Reble, S. Lankes, F. Zeitz, and T. Bemmerl. Evaluation of hardware synchronization support of the scc many-core processor. In *Proceedings of the 4th USENIX Workshop on Hot Topics in Parallelism (HotPar 12), Berkeley, CA, USA*, 2012.

[21] S. Rhoads. Plasma-most mips i (tm) opcodes: overview. *Internet: http://opencores. org/project, plasma [May 2, 2012]*, 2006.